ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
**KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH**


## CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT - CO2003

---

## BÀI TẬP LỚN 2

# HIỆN THỰC CƠ CHẾ TÌM KIẾM TRONG VECTORSTORE SỬ DỤNG CẤU TRÚC DỮ LIỆU CÂY

---

TP. HỒ CHÍ MINH, THÁNG 11/2025

# ĐẶC TẢ BÀI TẬP LỚN

**Phiên bản 1.0**

## 1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên sẽ có khả năng:

- Thành thạo lập trình hướng đối tượng (OOP).
- Phát triển các cấu trúc dữ liệu dạng cây.
- Sử dụng các cấu trúc dữ liệu dạng cây để hiện thực một VectorStore có hỗ trợ các loại cơ chế tìm kiếm.

## 2 Dẫn nhập

Trong Bài tập lớn 2, sinh viên được yêu cầu hiện thực một VectorStore sử dụng các cấu trúc dữ liệu dạng cây để lưu trữ và truy vấn hiệu quả các vector đa chiều. Cụ thể, cây AVL được dùng làm chỉ mục chính, sắp xếp các bản ghi theo khoảng cách Euclid đến một reference vector nhằm bảo đảm tính cân bằng và hỗ trợ các thao tác chèn/tìm/xóa với độ phức tạp logarit, đồng thời cho phép duyệt theo trật tự khoảng cách để phục vụ các truy vấn phạm vi. Bên cạnh đó, hệ thống bổ sung một chỉ mục phụ bằng red–black tree (RBT) sắp theo chuẩn (norm) của vector, đóng vai trò "bộ lọc tiền xử lý" giúp nhanh chóng chọn ra tập ứng viên gần về độ lớn, từ đó giảm số phép tính khoảng cách thật khi tìm top-k láng giềng gần nhất. Thiết kế hai chỉ mục phối hợp này phản ánh một mô hình VectorStore đơn giản nhưng hiệu quả, tương thích với các bài toán tìm kiếm ngữ nghĩa, gợi ý nội dung và quản lý tri thức dựa trên embedding.

Mục tiêu của bài tập là giúp sinh viên:

- Củng cố kỹ năng lập trình hướng đối tượng thông qua việc thiết kế các lớp cây và bản ghi vector có trách nhiệm rõ ràng.
- Hiểu và hiện thực chi tiết hai cấu trúc cây cân bằng (AVL, RBT), bao gồm quy tắc màu/chiều cao, các phép quay, và quy trình tái cân bằng khi chèn/xóa.
- Áp dụng cấu trúc cây vào một VectorStore thực tiễn: dùng AVL làm chỉ mục chính theo khoảng cách đến reference vector, và dùng RBT làm chỉ mục norm để lọc phạm vi trước khi xếp hạng.

# 3 Mô tả

## 3.1 Cây AVL

Cây AVL là một cây tìm kiếm nhị phân tự cân bằng, trong đó độ chênh lệch chiều cao giữa cây con trái và cây con phải của mọi nút (gọi là hệ số cân bằng) không vượt quá 1. Cây AVL đảm bảo các thao tác tìm kiếm, chèn và xóa có độ phức tạp $O(\log n)$ trong mọi trường hợp.

### 3.1.1 Lớp AVLNode

Lớp `AVLNode<K, T>` mô tả một nút trong cây AVL. Mỗi node lưu trữ:

#### 3.1.1.1 Các thuộc tính:

- `K key`: khoá dùng để so sánh (kiểu template `K`).
- `T data`: Giá trị dữ liệu được lưu tại node (kiểu template `T`).
- `AVLNode* pLeft`: Con trỏ tới node con trái.
- `AVLNode* pRight`: Con trỏ tới node con phải.
- `BalanceValue balance`: Hệ số cân bằng của node, được biểu diễn bằng enum:

| Tên | Giá trị | Ý nghĩa |
|---|---|---|
| `LH` | -1 | Left Higher (cây con trái cao hơn) |
| `EH` | 0 | Equal Height (hai cây con bằng nhau) |
| `RH` | 1 | Right Higher (cây con phải cao hơn) |

#### 3.1.1.2 Phương thức:

- `AVLNode(const K& key, const T& value)`: Constructor khởi tạo node với giá trị `key`, `value`, các con trỏ `pLeft` và `pRight` được gán `nullptr`, và `balance` được đặt là `EH`.
  - **Độ phức tạp**: $O(1)$.

### 3.1.2 Lớp AVLTree

#### 3.1.2.1 Thuộc tính:

- `AVLNode<K, T>* root`: Con trỏ tới node gốc của cây AVL. Được khởi tạo là `nullptr` khi tạo cây rỗng.

#### 3.1.2.2 Phương thức protected (hỗ trợ nội bộ):

- `AVLNode<K, T>* rotateRight(AVLNode<K, T>*& node)`
  Thực hiện phép xoay phải tại node để cân bằng cây khi cây con trái cao hơn cây con phải quá 1.
  - **Đầu vào**: Node cần được xoay phải.
  - **Đầu ra**: Node mới làm gốc sau khi xoay phải.
  - **Độ phức tạp**: $O(1)$.
- `AVLNode<K, T>* rotateLeft(AVLNode<K, T>*& node)`
  Thực hiện phép xoay trái tại node để cân bằng cây khi cây con phải cao hơn cây con trái quá 1.
  - **Đầu vào**: Node cần được xoay trái.
  - **Đầu ra**: Node mới làm gốc sau khi xoay trái.
  - **Độ phức tạp**: $O(1)$.

#### 3.1.2.3 Phương thức public:

- `AVLTree()`
  Constructor khởi tạo một cây AVL rỗng với `root = nullptr`.
  - **Độ phức tạp**: $O(1)$.
- `~AVLTree()`
  Destructor giải phóng toàn bộ bộ nhớ được cấp phát cho cây (tất cả các node).
  - **Độ phức tạp**: $O(n)$.
- `void insert(const K& key, const T& value)`
  Chèn một node vào cây AVL. Phương thức tự động duy trì tính cân bằng của cây.
  - **Đầu vào**: Khoá `key` và giá trị cần chèn `value`.
  - **Độ phức tạp**: $O(\log n)$.
  - **Lưu ý**: Nếu giá trị đã tồn tại, không thực hiện chèn.
- `void remove(const K& key)`
  Xóa một giá trị khỏi cây AVL. Phương thức tự động tái cân bằng cây nếu cần. Sử dụng phương pháp chọn nút có khoá bé nhất của cây con bên phải để thay thế.
  - **Đầu vào**: Khoá `key` cần xoá ra khỏi cây
  - **Độ phức tạp**: $O(\log n)$.

  - **Lưu ý**: Xoá nút có khoá `key` đầu tiên nếu tồn tại. Nếu giá trị không tồn tại, không thực hiện gì.

- `bool contains(const T& key) const`
  Kiểm tra xem một khoá có tồn tại trong cây hay không.
  - **Đầu vào**: Khoá cần kiểm tra `key`.
  - **Đầu ra**: `true` nếu giá trị tồn tại, `false` nếu không.
  - **Độ phức tạp**: $O(\log n)$.

- `int getHeight() const`
  Trả về chiều cao của cây AVL.
  - **Đầu ra**: Chiều cao của cây. Trả về 0 nếu cây rỗng.
  - **Độ phức tạp**: $O(n)$.

- `int getSize() const`
  Trả về số lượng node hiện có trong cây.
  - **Đầu ra**: Số node trong cây.
  - **Độ phức tạp**: $O(n)$.

- `bool empty() const`
  Kiểm tra xem cây có rỗng hay không.
  - **Đầu ra**: `true` nếu cây rỗng, `false` nếu cây có ít nhất một node.
  - **Độ phức tạp**: $O(1)$.

- `void clear()`
  Xóa toàn bộ các node trong cây và đặt `root = nullptr`. Giải phóng tất cả bộ nhớ.
  - **Độ phức tạp**: $O(n)$.

- `void printTreeStructure() const`
  In cấu trúc cây theo từng mức (level-order traversal) để trực quan hóa cây AVL (được cung cấp sẵn)
  - **Độ phức tạp**: $O(n)$.

- `void inorderTraversal(void (*action)(const T&)) const`
  Duyệt cây theo thứ tự in-order (trái-gốc-phải) và thực thi hàm `action` trên mỗi giá trị.
  - **Đầu vào**: Con trỏ tới hàm `action` để xử lý từng phần tử.
  - **Độ phức tạp**: $O(n)$.

## 3.2 Red-Black Tree

### 3.2.1 Mô tả

Red-black tree là một cây tìm kiếm nhị phân tự cân bằng, trong đó mỗi nút mang một thuộc tính màu (đỏ hoặc đen) nhằm khống chế độ mất cân bằng của cây, vì vậy chiều cao của cây tỉ lệ theo $\log n$ và các phép toán tra cứu đạt thời gian $O(\log n)$.

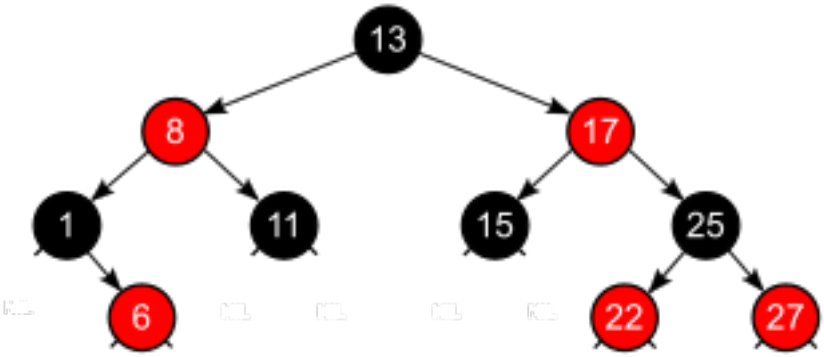
Red-black tree duy trì bất biến của cây tìm kiếm nhị phân: với mọi nút $x$, mọi khóa trong cây con trái của $x$ nhỏ hơn khóa của $x$, và mọi khóa trong cây con phải của $x$ lớn hơn khóa của $x$.

**Quy tắc màu:** Gọi `NIL` là các lá rỗng (được coi là nút đen) và `root` là gốc cây. Một red-black tree hợp lệ phải thỏa đồng thời các quy tắc sau:

- Mỗi nút được tô màu hoặc đỏ hoặc đen.
- Nút gốc luôn là đen.
- Mọi lá `NIL` được coi là đen.
- Nếu một nút là đỏ thì cả hai con của nó đều là đen.
- Với mọi nút, mọi đường đơn từ nút đó tới các lá `NIL` bên dưới đều đi qua cùng một số lượng nút đen. Số lượng nút đen này được gọi là **black-hight**

Từ các quy tắc trên suy ra chiều cao của red-black tree bị chặn bởi một hằng số nhân của $\log n$; do đó các phép *search* và các thao tác cập nhật (sau khi cân bằng lại) đều có độ phức tạp trường hợp xấu nhất $O(\log n)$.

Sinh viên có thể đọc tên các tài liệu về red-black tree tại đây: 1, 2, 3




Hình 1: Red-Black Tree

### 3.2.2 Quy tắc

**3.2.2.1 Chèn và tái cân bằng** Khi chèn một khóa mới vào red-black tree, đầu tiên chèn như trong cây tìm kiếm nhị phân và tô màu nút mới là đỏ; sau đó xoay các nút và thay đổi màu để đảm bảo quy tắc màu. Gọi $N$ là nút vừa chèn, $P = \text{parent}(N)$ là parent, $G = \text{parent}(P)$ là grand-parent, và $U$ là "uncle" của $N$ (anh/em của $P$). Các bước xử lý:

1. **Trường hợp 1 (gốc):** Nếu $N$ là gốc, tô $N$ thành đen và kết thúc. Trường hợp này đảm bảo quy tắc "gốc đen".
2. **Trường hợp 2 (parent đỏ, uncle đỏ – recolor):** Nếu $P$ là đỏ và $U$ tồn tại và cũng đỏ, thực hiện đổi màu: tô $P$ và $U$ thành đen, tô $G$ thành đỏ. Đặt nút $N$ thành $G$ và lặp lại quá trình sửa chữa vi phạm tại nút $N$ mới để đảm bảo quy tắc màu của toàn bộ cây.
3. **Trường hợp 3 (tam giác – quay trước rồi đổi vai):**
   - Nếu $P$ là đỏ, $U$ đen (hoặc không tồn tại), và cấu hình là "trái–phải" ($N$ là con phải của $P$, $P$ là con trái của $G$), thực hiện *xoay trái* tại $P$; sau quay, đặt nút $N$ thành $P$, để đưa về cấu hình "đường thẳng" và tiếp tục quá trình sửa chữa vi phạm.
   - Nếu cấu hình là "phải–trái" ($N$ là con trái của $P$, $P$ là con phải của $G$), thực hiện *xoay phải* tại $P$; sau đó đặt nút $N$ thành $P$, để đưa về cấu hình "đường thẳng" và tiếp tục quá trình sửa chữa vi phạm.
4. **Trường hợp 4 (đường thẳng – quay tại grand-parent rồi recolor):**
   - Nếu sau bước trước (hoặc ngay từ đầu) cấu hình là "trái–trái" ($N$ là con trái của $P$, $P$ là con trái của $G$), thực hiện *xoay phải* tại $G$.
   - Nếu cấu hình là "phải–phải" ($N$ là con phải của $P$, $P$ là con phải của $G$), thực hiện *xoay trái* tại $G$.

   Sau phép quay tại $G$: tô $P$ (nút lên làm parent sau quay) thành **đen** và tô $G$ thành **đỏ**. Khi đó mọi quy tắc màu được khôi phục và kết thúc.

Sinh viên có thể tham khảo thêm các ví dụ minh hoạt tại đây: 1, 2

**3.2.2.2 Xoá và tái cân bằng** Xoá một nút $Z$ theo hai bước: (i) nếu $Z$ có hai con, hoán đổi dữ liệu của $Z$ với *tiền nhiệm* của $Z$ (nút lớn nhất trong cây con trái), rồi đặt $Z$ thành nút tiền nhiệm đó; (ii) xoá $Z$ khi $Z$ có nhiều nhất một con khác `NIL`. Gọi $X$ là nút con (có thể là `NIL`) được "kéo lên" thay vị trí $Z$; nếu nút bị loại bỏ là đỏ thì kết thúc, nếu là đen thì phát sinh "thiếu đen" trên $X$ cần sửa bằng các trường hợp sau (ký hiệu $P = \text{parent}(X)$, $S$ là anh/em của $X$, $S_L, S_R$ là hai con của $S$):

1. **Gốc:** Nếu $X$ trở thành gốc, tô $X$ đen và kết thúc.
2. **Anh/em đỏ (xoay cha, đổi màu):** Nếu $S$ đỏ, xoay quanh $P$ để đưa $S$ lên làm cha mới của $P$, rồi đổi màu $S$ thành đen và $P$ thành đỏ. Khi đó anh/em mới của $X$ là đen; chuyển sang xét các trường hợp 3–5.
3. **Anh/em đen, cả hai con của $S$ đen (đổi màu, đẩy thiếu đen lên):** Tô $S$ đỏ, đặt $X \leftarrow P$ và lặp lại từ đầu ở mức cao hơn. Nếu $P$ đỏ, đổi $P$ thành đen và kết thúc.
4. **Anh/em đen, "con gần" đỏ, "con xa" đen (xoay tại $S$):**
   - Nếu $X$ là con trái của $P$ và $S_L$ đỏ, $S_R$ đen: xoay phải tại $S$, tô $S$ đỏ và $S_L$ đen; chuyển sang trường hợp 5.
   - Đối xứng: nếu $X$ là con phải của $P$ và $S_R$ đỏ, $S_L$ đen: xoay trái tại $S$, tô $S$ đỏ và $S_R$ đen; chuyển sang trường hợp 5.
5. **Anh/em đen, "con xa" đỏ (xoay tại $P$, đổi màu kết thúc):**
   - Nếu $X$ là con trái của $P$ và $S_R$ đỏ: xoay trái tại $P$; tô $S$ nhận màu cũ của $P$, tô $P$ đen, tô $S_R$ đen; kết thúc.
   - Đối xứng: nếu $X$ là con phải của $P$ và $S_L$ đỏ: xoay phải tại $P$; tô $S$ nhận màu cũ của $P$, tô $P$ đen, tô $S_L$ đen; kết thúc.

#### 3.2.2.3 Tìm kiếm một nút trong red-black tree

Tìm kiếm một nút trong red-black tree cũng tương tự với việc tìm kiếm một nút trong cây nhị phân tìm kiếm.

### 3.2.3 Lớp RBTNode

#### 3.2.3.1 Thuộc tính

Một `RBTNode` biểu diễn một nút của red-black tree với các trường cơ bản sau:

- `K key`: khoá dùng để so sánh (kiểu template K)
- `T data`: dữ liệu được lưu tại node (kiểu template T)
- `Color color`: màu của nút, được kiểu diễn bằng kiểu enum.
- `RBTNode* parent`: con trỏ đến nút cha.
- `RBTNode* left`, `RBTNode* right`: con trỏ đến cây con trái/phải.

#### 3.2.3.2 Các phương thức

- `RBTNode(const K& key, const T& value)`: Constructor tạo nút mới với giá trị `key`, `value` và gán `color = RED`, `left = right = parent = nullptr`

  - Độ phức tạp: $O(1)$

- `void recolorToRed()`: Đổi màu của nút hiện tại sang `RED`
  - Độ phức tạp: $O(1)$
- `void recolorToBlack()`: Đổi màu của nút hiện tại sang `Black`
  - Độ phức tạp: $O(1)$

#### 3.2.4 Lớp RedBlackTree

##### 3.2.4.1 Thuộc tính

- `RBTNode<K,T>* root`: Con trỏ tới nút gốc của red–black tree. Khởi tạo `nullptr` khi cây rỗng.

##### 3.2.4.2 Các phương thức protected

- `void rotateLeft(RBTNode<K,T>* node)`
  Xoay trái tại nút `node` để cân bằng cây.
  - Đầu vào: `node` là nút cần xoay trái.
  - Đầu ra: Không có.
  - Độ phức tạp: $O(1)$.
  - Lưu ý: Bảo toàn bất biến BST; chỉ thay đổi cấu trúc cục bộ.
- `void rotateRight(RBTNode<K,T>* node)`
  Xoay phải tại nút `node` để tái cấu trúc liên kết cục bộ.
  - Đầu vào: `node` là nút cần xoay phải.
  - Đầu ra: Không có
  - Độ phức tạp: $O(1)$.
  - Lưu ý: Đối xứng với `rotateLeft`.
- `RBTNode<K,T>* lowerBoundNode(const K& key) const`
  Tìm nút đầu tiên có khoá $\geq$ `key`.
  - Đầu vào: `key`.
  - Đầu ra: Con trỏ nút thoả mãn hoặc `nullptr` nếu không có.
  - Độ phức tạp: $O(\log n)$.
- `RBTNode<K,T>* upperBoundNode(const K& key) const`
  Tìm nút đầu tiên có khoá $>$ `key`.

  - Đầu vào: `key`.
  - Đầu ra: Con trỏ nút thoả mãn hoặc `nullptr`/`NIL`.
  - Độ phức tạp: $O(\log n)$.

#### 3.2.4.3 Các phương thức public

- `RedBlackTree()`
  Constructor khởi tạo cây rỗng.
  - Độ phức tạp: $O(1)$.
- $\sim$ `RedBlackTree()`
  Destructor giải phóng toàn bộ nút người dùng cấp phát.
  - Đầu vào: Không có.
  - Độ phức tạp: $O(n)$.
- `bool empty() const`
  Kiểm tra cây rỗng.
  - Đầu ra: `true` nếu rỗng, `false` nếu ngược lại.
  - Độ phức tạp: $O(1)$.
- `int size() const`
  Trả về số nút hiện có.
  - Đầu vào: Không có.
  - Đầu ra: Số node trong cây.
  - Độ phức tạp: $O(n)$.
- `void clear()`
  Xoá toàn bộ cây, đặt `root` về rỗng.
  - Độ phức tạp: $O(n)$.
- `void insert(const K& key, const T& value)`
  Chèn cặp `(key, value)` vào cây. Nếu `key` đã tồn tại thì không làm gì cả.
  - Đầu vào: `key`, `value`.
  - Độ phức tạp: $O(\log n)$.
- `void remove(const K& key)`
  Xoá nút có khoá `key` đầu tiên nếu tồn tại. Phương thức tự động cân bằng, sử dụng phương pháp chọn nút có khoá lớn nhất của cây con bên trái để thay thế (nếu cần).
  - Đầu vào: `key`.

  - Độ phức tạp: $O(\log n)$.
  - Lưu ý: Nếu không tồn tại key, thì không trả thực hiện gì.

- `RBTNode<K, T>* find(const K& key) const`
  Tìm và trả về con trỏ tới nút đầu tiên có khoá `key`.
  - Đầu vào: `key`.
  - Đầu ra: `RBTNode<K, T>*` tới dữ liệu, hoặc `nullptr` nếu không có.
  - Độ phức tạp: $O(\log n)$.

- `bool contains(const K& key) const`
  Kiểm tra sự tồn tại của khoá.
  - Đầu vào: `key`.
  - Đầu ra: `true`/`false`.
  - Độ phức tạp: $O(\log n)$.

- `RBTNode<K, T>* lowerBound(const K& key, bool& found) const`
  Trả về địa chỉ tới nút có khoá **nhỏ nhất** $\geq$ `key`.
  - Đầu vào: `key`.
  - Đầu ra: `found = true` và khoá cận dưới nếu tồn tại; `found = false` nếu không có. Địa chỉ tới nút có khoá nhỏ nhất thoả điều kiện hoặc `nullptr` nếu không có
  - Độ phức tạp: $O(\log n)$.

- `RBTNode<K, T>* upperBound(const K& key, bool& found) const`
  Trả về địa chỉ tới nút có khoá **nhỏ nhất** $>$ `key`.
  - Đầu vào: `key`.
  - Đầu ra: Địa chỉ `found = true` và khoá cận dưới nếu tồn tại; `found = false` nếu không có. Địa chỉ tới nút có khoá nhỏ nhất thoả điều kiện hoặc `nullptr` nếu không có.
  - Độ phức tạp: $O(\log n)$.

- `void printTreeStructure() const`
  In cấu trúc cây theo từng mức (level-order traversal) để trực quan hóa red-black tree (được cung cấp sẵn)
  - **Độ phức tạp**: $O(n)$.

## 3.3 Vector Store và cơ chế tìm kiếm trong VectorStore

### 3.3.1 Giới thiệu

VectorStore là cấu trúc dữ liệu chuyên biệt để lưu trữ và truy vấn các vector đa chiều; cây AVL được dùng làm chỉ mục chính, sắp xếp các bản ghi theo khoảng cách đến một reference vector nhằm bảo đảm thao tác chèn/tìm/xóa ổn định và hỗ trợ duyệt theo trật tự khoảng cách. Bổ sung, hệ thống tích hợp một red–black tree (RBT) làm "norm index" sắp theo "chuẩn Euclidean" của từng vector; khi truy vấn, RBT cung cấp phép lọc nhanh các ứng viên bằng cận dưới/cận trên trên trục chuẩn, chỉ giữ lại những vector có chuẩn nằm trong một cửa sổ quanh chuẩn của truy vấn trước khi tính điểm chính xác và chọn top-$k$, nhờ đó giảm đáng kể số phép tính và tăng tốc truy vấn láng giềng gần nhất trong các ứng dụng embedding AI/ML.

### 3.3.2 Kiến trúc và nguyên lý hoạt động

**3.3.2.1 Khái niệm VectorStore dựa trên khoảng cách từ tâm** VectorStore không sắp xếp vector dựa trên khóa id hay một chiều cụ thể, mà sử dụng khoảng cách Euclid đến một điểm tham chiếu được gọi là reference vector. Phương pháp này mang lại các lợi ích:

- Các vector gần nhau về mặt khoảng cách sẽ được lưu trữ gần nhau trong cây, tối ưu hóa truy vấn láng giềng gần nhất.
- Cho phép truy vấn phạm vi nhanh chóng bằng cách duyệt cây theo thứ tự khoảng cách.
- Hỗ trợ tìm kiếm xấp xỉ (approximate nearest neighbor search) bằng cách chỉ truy vấn các nhánh cây có khoảng cách phù hợp.
- Bổ sung chỉ mục phụ bằng cây đỏ–đen (RBT) sắp theo "chuẩn Euclid" của vector, cho phép lọc nhanh ứng viên bằng `lower_bound`/`upper_bound` trên trục [*chuẩn*] để lấy tập nhỏ quanh [*chuẩn*] của truy vấn trước khi tính điểm chính xác, từ đó giảm đáng kể số phép tính và tăng tốc top-k.

#### 3.3.2.2 Hai khái niệm vector quan trọng

- **Reference Vector**: Vector gốc được người dùng cung cấp lúc khởi tạo VectorStore. Đây là điểm tham chiếu để tính khoảng cách từ tất cả các vector khác. Reference vector có thể không nằm trong store.
- **Root Vector của AVL Tree**: Không phải là reference vector, mà là một vector **thuộc store** được chọn sao cho khoảng cách của nó đến reference vector gần nhất với **khoảng**

**cách trung bình** từ reference vector đến tất cả vector trong store. Root vector này được sử dụng làm nút gốc của cây AVL Tree.

### 3.3.3 Lớp VectorRecord

Mỗi vector được đóng gói thành một bản ghi để quản lý dữ liệu và metadata liên quan:

- `int id`: Định danh duy nhất cho vector (không được sử dụng làm khóa sắp xếp trong AVL). `id` tăng theo quy tắc: `id` của record mới nhất sẽ bằng `id` lớn nhất đang được lưu trong kho cộng lên 1 đơn vị.
- `string rawText`: Chuỗi văn bản gốc trước khi được ánh xạ thành vector.
- `int rawLength`: Độ dài của chuỗi văn bản gốc.
- `vector<float>* vector`: Con trỏ tới dữ liệu vector số thực.
- `double distanceFromReference`: Khoảng cách từ reference vector đến vector này (dùng để tính khoảng cách trung bình).
- `double norm`: "chuẩn Euclidean" của vector số thực. Dùng để xây dựng red-black tree theo chuẩn.

### 3.3.4 Lớp VectorStore

#### 3.3.4.1 Thuộc tính của lớp

- `AVLTree<double, VectorRecord>* vectorStore`: Cây AVL lưu trữ các bản ghi VectorRecord. Giá trị `distanceFromReference` đóng vai trò là khoá.
- `RedBlackTree<double, VectorRecord>* normIndex`: Red-black tree lưu trữ các bản ghi VectorRecord, đóng vai trò là một chỉ mục thứ cấp để hỗ trợ tìm kiếm. Giá trị `norm` đóng vai trò là khoá.
- `vector<float>* referenceVector`: Vector gốc được người dùng cung cấp lúc khởi tạo, dùng làm điểm tham chiếu để tính toán khoảng cách của tất cả vector khác. Reference vector không nhất thiết phải nằm trong store.
- `VectorRecord* rootVector`: Bản ghi của vector được chọn làm root vector của cây AVL (vector thuộc store có khoảng cách gần nhất với khoảng cách trung bình từ reference vector).
- `int dimension`: Số chiều cố định của mọi vector trong store.
- `int count`: Số lượng bản ghi vector hiện có trong store.
- `double averageDistance`: Khoảng cách trung bình từ reference vector đến tất cả vector hiện có trong store. Được dùng làm tiêu chí để tìm root vector.

- `vector<float>* (*embeddingFunction)(const string&)`: Con trỏ hàm ánh xạ từ chuỗi văn bản sang vector số thực nhiều chiều.

#### 3.3.4.2 Các phương thức công khai (public)

**Khởi tạo và quản lý cơ bản**

- `VectorStore(int dimension = 512, vector<float>* (*embeddingFunction)(const string&), const vector<float>& referenceVector)`
  Khởi tạo một VectorStore rỗng với số chiều cố định, hàm nhúng, và reference vector. Store sẽ trống cho đến khi thêm vector đầu tiên.
  - **Ngoại lệ**: Không có.
  - **Độ phức tạp**: $O(1)$.
- `~VectorStore()`
  Destructor giải phóng toàn bộ bộ nhớ được cấp phát cho cây AVL Tree, các bản ghi vector và các dữ liệu liên quan.
  - **Độ phức tạp**: $O(n)$.
- `int size()`
  Trả về số lượng vector hiện có trong kho.
  - **Độ phức tạp**: $O(1)$.
- `bool empty()`
  Kiểm tra kho có rỗng hay không.
  - **Độ phức tạp**: $O(1)$.
- `void clear()`
  Xóa toàn bộ vector và siêu dữ liệu đi kèm khỏi cây AVL và red-black tree, nhưng giữ lại reference vector.
  - **Độ phức tạp**: $O(n)$.

**Tiền xử lý và quản lý dữ liệu**

- `vector<float>* preprocessing(string rawText)`
  Tiền xử lý văn bản đầu vào thành vector số thực. Gọi hàm embedding để ánh xạ văn bản, sau đó chuẩn hóa chiều của vector (cắt bớt hoặc thêm padding).

  - **Yêu cầu**:
    1. Gọi `embeddingFunction` để ánh xạ chuỗi `rawText` thành vector.
    2. Nếu số chiều lớn hơn `dimension`, cắt bớt các phần tử ở cuối.
    3. Nếu số chiều nhỏ hơn `dimension`, thêm các phần tử giá trị 0.0 vào cuối (post-padding).
  - **Độ phức tạp**: $O(d)$, với $d$ là số chiều.

- `void addText(string rawText)`
  Thêm một bản ghi mới vào kho từ chuỗi văn bản gốc. Phương thức sẽ:
  1. Sử dụng preprocessing để chuyển đổi thành vector.
  2. Tính khoảng cách từ reference vector đến vector vừa tạo.
  3. Cập nhật khoảng cách trung bình.
  4. Tính "chuẩn Euclidean" của vector.
  5. Nếu store rỗng, đặt vector này làm root vector.
  6. Nếu store không rỗng:
     - Chèn vector vào cây AVL. Nếu khoảng cách của vector mới gần với khoảng cách trung bình hơn root hiện tại, tái cấu trúc cây với root vector mới.
     - Chèn vector vào red-black tree, đảm bảo tính cân bằng của cây sau khi chèn.
  - **Độ phức tạp**: $O(\log n)$ trường hợp bình thường, $O(n \log n)$ nếu cần thay đổi root.

- `VectoRecord* getVector(int index)`
  Truy xuất tham chiếu tới vector tại vị trí `index` khi duyệt **cây AVL** theo thứ tự in-order (tính từ 0).
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: $O(n)$.

- `string getRawText(int index)`
  Lấy chuỗi văn bản gốc tại vị trí `index` trong duyệt **cây AVL** theo thứ tự in-order
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: $O(n)$.

- `int getId(int index)`
  Lấy định danh (id) của vector tại vị trí `index` trong duyệt **cây AVL** theo thứ tự in-order.
  - **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
  - **Độ phức tạp**: $O(n)$.

- `bool removeAt(int index)`

Xóa vector và metadata tại vị trí `index` trong duyệt **cây AVL** theo thứ tự in-order, giải phóng bộ nhớ và xóa khỏi cây AVL. Nếu vector bị xóa là root vector, tìm vector mới gần nhất với khoảng cách trung bình cập nhật để làm root mới. Đồng thời, vector đó cũng phải được xoá trong red-black tree để đảm bảo tính thống nhất.

- **Ngoại lệ**: Ném `out_of_range("Index is invalid!")` nếu chỉ số không hợp lệ.
- **Độ phức tạp**: $O(n + \log n) = O(n)$.

### Quản lý reference vector và hàm nhúng

- `void setReferenceVector(const vector<float>& newReference)`
  Đổi reference vector sang vector mới. Phương thức sẽ tái tính toán khoảng cách từ reference vector mới đến tất cả vector hiện có, cập nhật khoảng cách trung bình, tìm vector mới gần nhất với khoảng cách trung bình để làm root vector mới, và tái cấu trúc cây AVL.
  **Quy trình tái cấu trúc cây AVL**
  - **Bước 1: Tính khoảng cách.** Với mỗi vector trong kho hiện tại, tính khoảng cách của nó đến *reference vector*.
  - **Bước 2: Xác định nút gốc (root vector).** Tính khoảng cách trung bình của tất cả các vector đến *reference vector*. Chọn *root vector* là vector có khoảng cách gần nhất với giá trị trung bình này.
  - **Bước 3: Sắp xếp theo thứ tự tăng dần.** Sắp xếp toàn bộ các vector theo thứ tự tăng dần của khoảng cách đến *reference vector*. Thứ tự này tương ứng với thứ tự duyệt *in-order* của cây AVL sau khi tái cấu trúc.
  - **Bước 4: Xây dựng lại cây AVL.**
    1. Chia danh sách đã sắp xếp thành hai nửa: nửa bên trái gồm các vector có khoảng cách nhỏ hơn *root vector*, và nửa bên phải gồm các vector có khoảng cách lớn hơn.
    2. Trong mỗi nửa cây, chọn phần tử giữa làm gốc, sau đó đệ quy cho hai phần còn lại.
    3. Gắn hai cây con vừa xây dựng vào hai nhánh trái và phải của *root vector* để tạo thành cây AVL hoàn chỉnh
  - **Độ phức tạp**: $O(n \log n)$.
- `VectorRecord* getReferenceVector() const`
  Trả về con trỏ đến reference vector hiện tại (được người dùng truyền vào).

  - **Độ phức tạp**: $O(1)$.

- `VectorRecord* getRootVector() const`
  Trả về con trỏ tới root vector hiện tại của cây AVL
  - **Độ phức tạp**: $O(1)$.
- `double getAverageDistance() const`
  Trả về khoảng cách trung bình từ reference vector đến tất cả vector trong store.
  - **Độ phức tạp**: $O(1)$.
- `void setEmbeddingFunction(vector<float>* (*newEmbeddingFunction)(const string&))`
  Cập nhật hàm ánh xạ văn bản sang vector.
  - **Độ phức tạp**: $O(1)$.

### Duyệt và lặp

- `void forEach(void (*action)(vector<float>&, int, string&))`
  Duyệt toàn bộ bản ghi trong AVL Tree theo thứ tự in-order (khoảng cách từ reference vector tăng dần) và thực thi hàm `action` trên mỗi bản ghi.
  - **Độ phức tạp**: $O(n)$.
- `std::vector<int> getAllIdsSortedByDistance() const`
  Trả về danh sách tất cả các id của bản ghi vector sắp xếp theo khoảng cách tăng dần từ reference vector của cây AVL.
  - **Đầu ra**: Vector chứa các id.
  - **Độ phức tạp**: $O(n)$.
- `std::vector<VectorRecord*> getAllVectorsSortedByDistance() const`
  Trả về danh sách tất cả các bản ghi vector sắp xếp theo khoảng cách tăng dần từ reference vector của cây AVL.
  - **Đầu ra**: Vector chứa các con trỏ bản ghi.
  - **Độ phức tạp**: $O(n)$.

### Các độ đo khoảng cách

- `double cosineSimilarity(const vector<float>& v1, const vector<float>& v2)`
  Tính độ tương đồng cosine giữa hai vector theo công thức:

$$\cos(\theta) = \frac{A \cdot B}{\|A\| \cdot \|B\|}$$

  - **Độ phức tạp**: $O(d)$.

- `double l1Distance(const vector<float>& v1, const vector<float>& v2)`
  Tính khoảng cách Manhattan giữa hai vector:

$$d(\vec{A}, \vec{B}) = \sum_{i=1}^{d} |A_i - B_i|$$

  - **Độ phức tạp**: $O(d)$.

- `double l2Distance(const vector<float>& v1, const vector<float>& v2)`
  Tính khoảng cách Euclidean giữa hai vector:

$$d(\vec{A}, \vec{B}) = \sqrt{\sum_{i=1}^{d} (A_i - B_i)^2}$$

  - **Độ phức tạp**: $O(d)$.

### Ước lượng ngưỡng `D` từ `k`

- `double estimateD_Linear(const vector<float>& query, int k, double averageDistance, const vector<float> reference, double c0_bias = 1e-9, double c1_slope = 0.05)`
  Ước lượng bán kính $D$ theo mô hình tuyến tính theo $k$ (Linear-in-$k$) dựa trên thống kê sẵn có của VectorStore.
  - **Đầu vào:**
    - `query`: vector truy vấn.
    - `k`: số láng giềng cần tìm.
    - `averageDistance`: $a = \text{mean}\{\|v - r\|\}$ đang được lưu trong kho.
    - `reference`: vector tham chiếu $r$.
    - `c0_bias`, `c1_slope`: hệ số tinh chỉnh đơn giản (mặc định 1e−9 và 0.05).
  - **Đầu ra:** Giá trị $D > 0$
  - **Công thức:** $d_r = \|\,\texttt{query} - \texttt{reference}\,\|\,, \qquad D = \big|\,d_r - \texttt{averageDistance}\,\big| + \texttt{c1\_slope} \cdot \texttt{averageDistance} \cdot k + \texttt{c0\_bias}\,.$

### Tìm kiếm láng giềng gần nhất

- `int findNearest(const vector<float>& query, string metric = "cosine")`

Tìm VectorRecord gần nhất với vector truy vấn theo độ đo metric. Hỗ trợ các metric: `cosine`, `euclidean`, `manhattan`.

- **Đầu ra**: Trả về id của VectorRecord gần nhất.
- **Ngoại lệ**: Ném `invalid_metric()` nếu metric không hợp lệ.
- **Độ phức tạp**: $O(n \times d)$.

`int* topKNearest(const vector<float>& query, int k, string metric = "cosine")`

Tìm $k$ VectorRecord gần nhất với vector truy vấn và trả về mảng động các `id`. Phương thức ứng dụng chỉ mục red-black tree theo chuẩn (norm) để lọc phạm vi trước, sau đó mới tính khoảng cách thật và xếp hạng.

- **Đầu ra:** Con trỏ mảng động `int` chứa `k` `id` theo thứ tự khoảng cách tăng dần, kích thước đúng `k` nếu có đủ ứng viên; nếu ít hơn `k` ứng viên, trả về số lượng thực lấy được.
- **Ngoại lệ:** Ném `invalid_metric()` nếu `metric` không hợp lệ; ném `invalid_k_value()` nếu `k` $\leq 0$ hoặc `k` quá lớn so với kích thước kho.
- **Độ phức tạp:** $O(\log n + m \cdot d + m \log k)$, trong đó $n$ là số vector, $d$ là số chiều, $m$ là số ứng viên sau khi lọc bằng red-black tree (kỳ vọng $m \ll n$).
- **Các bước thực hiện**:
  - **Bước 1**: Tính chuẩn Euclidean của truy vấn. Tính $n_q = \|q\|$.
  - **Bước 2**: Ước lượng bán kính $D$ từ $k$ bằng cách sử dụng phương thức `estimateD_Linear`
  - **Bước 3**: Lọc bằng red-black tree (normIndex). Dùng `lower_bound`( $n_q - D$ ) và `upper_bound`( $n_q + D$ ) trên red-black tree để thu dải ứng viên theo chuẩn, tạo tập ứng viên kích thước $m$.
    In ra giá trị `m` theo format: `"Value m: <m>"`
  - **Bước 4**: Tính khoảng các thật trên ứng viên. Với từng ứng viên trong dải, tính khoảng cách thật theo `metric` đã chọn.
  - **Bước 5**: Chọn top-$k$ và trả về một mảng động `int` kích thước `k` theo thứ tự khoảng cách tăng dần.

### Truy vấn phạm vi

- `int* rangeQueryFromRoot(double minDist, double maxDist) const`
  Tìm tất cả các vector có khoảng cách từ root vector của cây AVL nằm trong phạm vi $[minDist, maxDist]$.
  - **Đầu ra**: Danh sách id của các vector.
  - **Độ phức tạp**: $O(k)$ với $k$ là số vector trong phạm vi.

- `int* rangeQuery(const vector<float>& query, double radius, string metric = "cosine") const`
  Tìm tất cả các vector trên cây AVL có khoảng cách/độ tương đồng từ vector truy vấn nằm trong bán kính theo metric.
  - **Đầu ra**: Danh sách id của các vector thỏa mãn.
  - **Ngoại lệ**: Ném `invalid_metric()` nếu metric không hợp lệ.
  - **Độ phức tạp**: $O(n \times d)$.
- `int* boundingBoxQuery(const vector<float>& minBound, const vector<float>& maxBound) const`
  Tìm tất cả các vector trên cây AVL nằm trong hộp giới hạn được định nghĩa bởi hai điểm minBound và maxBound.
  - **Đầu ra**: Danh sách id các vector thỏa mãn.
  - **Độ phức tạp**: $O(n)$.

**Các phương thức hỗ trợ nâng cao**

- `double getMaxDistance() const`
  Tìm khoảng cách lớn nhất từ reference vector tới bất kỳ vector nào trong store.
  - **Độ phức tạp**: $O(n)$.
- `double getMinDistance() const` Tìm khoảng cách nhỏ nhất từ reference vector.
  - **Độ phức tạp**: $O(\log n)$.
- `VectorRecord computeCentroid(const std::vector<VectorRecord*>& records) const`
  Tính centroid (tâm) của tập các vector theo công thức trung bình từng chiều.
  - **Độ phức tạp**: $O(m \times d)$ với $m$ là số vector, $d$ là chiều.
- `VectorRecord* findVectorNearestToDistance(double targetDistance) const`
  Tìm vector trong store có khoảng cách từ reference vector gần nhất với giá trị mục tiêu `targetDistance`.
  - **Độ phức tạp**: $O(n)$.

## 3.4 Cơ chế xác định Root Vector

### 3.4.1 Thuật toán tìm Root Vector

1. Tính khoảng cách từ reference vector đến mỗi vector trong store.

2. Tính khoảng cách trung bình:

$$\text{avgDist} = \frac{1}{n}\sum_{i=1}^{n}\text{distance}_i$$

3. Tìm vector có khoảng cách gần nhất với avgDist:

$$\text{rootVector} = \arg\min_i |\text{distance}_i - \text{avgDist}|$$

4. Đặt vector tìm được làm root vector của cây AVL.

### 3.4.2 Ví dụ minh họa

Giả sử VectorStore với reference vector ref $= (0, 0, 0)$ có 5 vector:

$$v_1 = (1.0, 0, 0), \quad d_1 = 1.0 \tag{1}$$
$$v_2 = (2.0, 0, 0), \quad d_2 = 2.0 \tag{2}$$
$$v_3 = (3.0, 0, 0), \quad d_3 = 3.0 \tag{3}$$
$$v_4 = (4.0, 0, 0), \quad d_4 = 4.0 \tag{4}$$
$$v_5 = (5.0, 0, 0), \quad d_5 = 5.0 \tag{5}$$

Khoảng cách trung bình:

$$\text{avgDist} = \frac{1+2+3+4+5}{5} = 3.0$$

Vector có khoảng cách gần nhất với 3.0 là $v_3$ (vì $|3.0 - 3.0| = 0$). Do đó, $v_3$ được chọn làm root vector của cây AVL.

Cây AVL sẽ được xây dựng với $v_3$ làm root, và các vector khác sắp xếp theo khoảng cách từ $v_3$:

- $v_1, v_2$: Cây con trái (khoảng cách nhỏ hơn từ $v_3$).
- $v_4, v_5$: Cây con phải (khoảng cách lớn hơn từ $v_3$).

### 3.5 Kết luận

VectorStore với cây AVL Tree cung cấp một cách hiệu quả và tối ưu để lưu trữ, tìm kiếm, và truy vấn dữ liệu vector đa chiều. Bằng cách chọn root vector thông minh (gần nhất với khoảng cách trung bình), hệ thống tự động tối ưu cấu trúc cây để phục vụ tốt nhất cho các truy vấn trong không gian vector. Cây AVL đảm bảo độ phức tạp logarit cho các thao tác cơ bản, trong khi duyệt theo khoảng cách từ root cho phép tối ưu hóa phạm vi truy vấn và hỗ trợ tìm kiếm xấp xỉ. Hệ thống tự động cập nhật root vector khi dữ liệu thay đổi, đảm bảo cấu trúc cây luôn cân bằng và tối ưu cho các ứng dụng xử lý vector (embedding) hiện đại.

## 4 Yêu cầu và chấm điểm

### 4.1 Yêu cầu

Để hoàn thành bài tập này, sinh viên cần:

1. Đọc toàn bộ file mô tả này.
2. Tải file **initial.zip** và giải nén. Sau khi giải nén, sinh viên sẽ nhận được các file bao gồm: `utils.h`, `main.cpp`, `main.h`, `VectorStore.h`, `VectorStore.cpp`. Sinh viên chỉ nộp 2 file, đó là `VectorStore.h` và `VectorStore.cpp`. Do đó, không được phép chỉnh sửa file `main.h` khi kiểm tra chương trình.
3. Sinh viên sử dụng lệnh sau để biên dịch:
   `g++ -o main main.cpp VectorStore.cpp -I . -std=c++17`
   Lệnh trên được sử dụng trong Command Prompt/Terminal để biên dịch chương trình. Nếu sinh viên sử dụng IDE để chạy chương trình, cần lưu ý: thêm tất cả các file vào project/workspace của IDE; chỉnh lại lệnh biên dịch trong IDE cho phù hợp. IDE thường cung cấp nút Build (biên dịch) và Run (chạy). Khi bấm Build, IDE sẽ chạy câu lệnh biên dịch tương ứng, thông thường chỉ biên dịch file `main.cpp`. Sinh viên cần tìm cách cấu hình để thay đổi câu lệnh biên dịch, cụ thể: thêm file `VectorStore.cpp`, thêm tùy chọn `-std=c++17`, và `-I .`
4. Chương trình sẽ được chấm trên nền tảng Unix. Môi trường của sinh viên và trình biên dịch có thể khác với môi trường chấm thực tế. Khu vực nộp bài trên LMS được thiết lập tương tự môi trường chấm thực tế. Sinh viên bắt buộc phải kiểm tra chương trình trên trang nộp bài, đồng thời sửa tất cả lỗi phát sinh trên hệ thống LMS để đảm bảo kết quả chính xác khi chấm cuối cùng.

5. Chỉnh sửa file `VectorStore.h` và `VectorStore.cpp` để hoàn thành bài tập, đồng thời đảm bảo hai yêu cầu sau:
   - Tất cả các phương thức được mô tả trong file hướng dẫn phải được cài đặt để chương trình có thể biên dịch thành công. Nếu sinh viên chưa hiện thực một phương thức nào đó, cần cung cấp phần hiện thực rỗng cho phương thức đó. Mỗi testcase sẽ gọi một số phương thức để kiểm tra kết quả trả về.
   - Trong file `VectorStore.h` chỉ được phép có đúng một dòng `#include "main.h"`, và trong file `VectorStore.cpp` chỉ được phép có một dòng `#include "VectorStore.h"`. Ngoài hai dòng này, không được phép thêm bất kỳ lệnh `#include` nào khác trong các file này.
   - Sinh viên không được sử dụng lệnh `#define TESTING` trong 2 file được yêu cầu chỉnh sửa.
6. Khuyến khích sinh viên được phép viết thêm các lớp, phương thức và thuộc tính phụ trợ trong các lớp yêu cầu hiện thực. Nhưng sinh viên phải đảm bảo các lớp, phương thức này không làm thay đổi yêu cầu của các phương thức được mô tả trong đề bài.
7. Sinh viên phải thiết kế và sử dụng các cấu trúc dữ liệu đã học.
8. Sinh viên bắt buộc phải giải phóng toàn bộ vùng nhớ cấp phát động khi chương trình kết thúc.

## 4.2 Thời hạn nộp bài

Thời hạn **theo thông báo trên LMS**. Sinh viên vui lòng nộp bài lên hệ thống trước thời hạn thông báo. Sinh viên tự chịu trách nhiệm nếu xuất hiện các lỗi trên hệ thống do nộp gần sát giờ quy định.

## 4.3 Chấm điểm

Toàn bộ mã nguồn của sinh viên sẽ được chấm trên bộ testcases ẩn, điểm được tính theo từng yêu cầu.

# 5 Harmony cho Bài tập lớn

Bài kiểm tra cuối kì của môn học sẽ có một số câu hỏi Harmony với nội dung của BTL.

Sinh viên phải giải quyết BTL bằng khả năng của chính mình. Nếu sinh viên gian lận trong BTL, sinh viên sẽ không thể trả lời câu hỏi Harmony và nhận điểm 0 cho BTL.

Sinh viên **phải** chú ý làm câu hỏi Harmony trong bài kiểm tra cuối kỳ. Các trường hợp không làm sẽ tính là 0 điểm cho BTL, và bị không đạt cho môn học. **Không chấp nhận giải thích và không có ngoại lệ.**

# 6 Quy định và xử lý gian lận

Bài tập lớn phải được sinh viên TỰ LÀM. Sinh viên sẽ bị coi là gian lận nếu:

- Có sự giống nhau bất thường giữa mã nguồn của các bài nộp. Trong trường hợp này, TẤT CẢ các bài nộp đều bị coi là gian lận. Do vậy sinh viên phải bảo vệ mã nguồn bài tập lớn của mình.
- Sinh viên không hiểu mã nguồn do chính mình viết, trừ những phần mã được cung cấp sẵn trong chương trình khởi tạo. Sinh viên có thể tham khảo từ bất kỳ nguồn tài liệu nào, tuy nhiên phải đảm bảo rằng mình hiểu rõ ý nghĩa của tất cả những dòng lệnh mà mình viết. Trong trường hợp không hiểu rõ mã nguồn của nơi mình tham khảo, sinh viên được đặc biệt cảnh báo là KHÔNG ĐƯỢC sử dụng mã nguồn này; thay vào đó nên sử dụng những gì đã được học để viết chương trình.
- Nộp nhầm bài của sinh viên khác trên tài khoản cá nhân của mình.
- Sinh viên sử dụng các công cụ AI trong quá trình làm bài tập lớn dẫn đến các mã nguồn giống nhau.

Trong trường hợp bị kết luận là gian lận, sinh viên sẽ bị điểm 0 cho toàn bộ môn học (không chỉ bài tập lớn).

**KHÔNG CHẤP NHẬN BẤT KỲ GIẢI THÍCH NÀO VÀ KHÔNG CÓ BẤT KỲ NGOẠI LỆ NÀO!**

Sau mỗi bài tập lớn được nộp, sẽ có một số sinh viên được gọi phỏng vấn ngẫu nhiên để chứng minh rằng bài tập lớn vừa được nộp là do chính mình làm.

Một số quy định khác:

- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài.
- Nội dung Bài tập lớn sẽ được Harmony với các câu hỏi trong bài kiểm tra cuối kì với nội dung tương tự.

# 7 Thay đổi so với phiên bản trước

- Cập nhật vector sử dụng cho các phương thức: **forEach**, **getAllIdsSortedByDistance**, **getAllVectorsSortedByDistance**, **getMaxDistance**, **getMinDistance**: Thay đổi root vector thành reference vector.

**————————HẾT————————**